Project 2 Report: RAID-6 Based Distributed Storage System

Remi Boukheloua – Amarildo Come – Elise Auvray

Abstract

This project report is a summary of the implementation process of a reliable distributed storage system based on RAID 6. Unlike RAID 5, it generates 2 parties in order to achieve this fault-tolerance level. The high-level objective of the project is to build system supporting storage and access of abstract "data objects" across storage nodes (disks) that can allow up-to 2 nodes failure. The system should include mechanisms to determine failure and carry out rebuilding the failed nodes. We use N folders as storage nodes assuming equal storage capacity. 2 of the nodes (N and N-1) are used as parity nodes. Files are segmented equally across the nodes and the parities are generated. Upon failure, a storage node missing, or files corrupted or missing, we are able to recover the missing node or file. We have finally measured to system performance to store, access and recover nodes.

1: Introduction

As part of the course on Advanced Topics in Distributed Systems, we modeled a distributed system based on RAID-6. Because of the rising demand for capacity, speed and reliability of storage system, a lot of research has been done on Redundant Array of Independent Disks (RAID). With RAID implementation, data blocks are distributed among multiple storage devices and can use one or more drives for fault-tolerance. There are different RAID levels for different data protection requirements.

Nowadays, RAID-5 is often deployed to protect the data. This system uses one parity drive in addition of the data so it can tolerate the loss of a single node. But, if two nodes fail at the same time, data will be lost. Thereby, RAID-6 (**figure 1**) is an improvement of RAID-5 because it makes it possible to tolerate the loss of two simultaneous node while having the possibility to recover the data. This new system, therefore, provides better data protection. This is thanks to the presence of two parity disks, called p and q. If you lose two data nodes, it is always possible to rebuild them using p, q and the remaining nodes. This innovation is very important because the probability of two node failing at the same time is quite high (**figure 2**). For example, a second drive can fail before the first one has been rebuilt, so, there are two nodes failing at the same time.

For this project, we built a distributed storage system to store and access data across storage nodes using RAID-6 to manage data losses. We have also implemented a mechanism to detect the storage node loss and finally we have implemented an algorithm to reconstruct the lost disk(s) and thus retrieve the data.

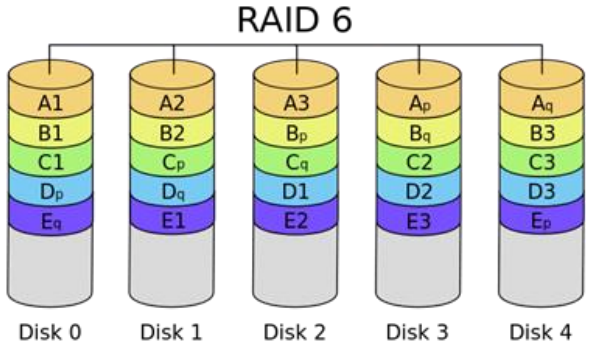

Figure 1: RAID 6 representation

2: RAID-6 Design

RAID 6 needs two parity nodes (p and q) and based on some algebra calculations and Boolean logic operations we are be able to generate p and q and to rebuild the lost data from the parity nodes.

2.1: RAID-6 architecture

To implement RAID-6, several nodes are need. There are 2 parity nodes, k data nodes and n total nodes. So, the number of parity node are 2 = n – k. It can thus correct up to two erasures. In addition, in RAID-6, the data files are split into bytes, store into the k data nodes, and then are used to calculate the two-parity nodes (p and q) store in their respective nodes. Therefore, because we use an 8 bit-symbol, the maximum code word length is $2^8 - 1 = 255$ so

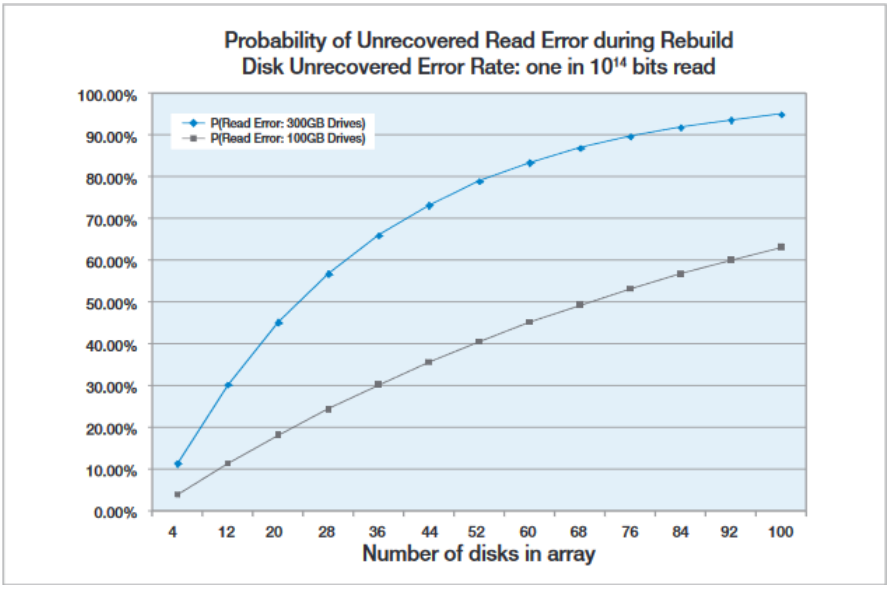

Figure 2: Latent Defects vs. Array Capacity, from [1] paper

the maximum number of nodes n would be 255, which means 253 data nodes and 2 arity drives.

2.1: Operations between nodes

To do operations between nodes, because it's bytes and note integer, the algebra rules are a little bit different. For example, the addition becomes an XOR operation. This operation as properties:

- Commutative: $A \oplus B = B \oplus A$; $A \otimes B = B \otimes A$;
- Associative: $(A \oplus B) \oplus C = A \oplus (B \oplus C)$; $(A \otimes B) \otimes C = A \otimes (B \otimes C)$;
- Distributive: $(A \oplus B) \otimes C = A \otimes C \oplus B \otimes C$;
- Inverse: $A^{-1} = 1 \div A$;
- Multiplicative Inverse : For all x in GF, there exists y in GF such that $x \otimes y = 1$
- Multiplicative Identity : $1 \otimes A = A$;
- Additive inverse : For all x in GF, there exists y in GF such that $x \oplus y = 0$.
- Additive Identity : For all x in GF, $x \oplus 0 = x$

2: Implementation of the system

2.1: Programming Environment

The system has been developed in Python 3.0. We used Jupyter notebook to be able to test our algorithms separately and have an easily readable presentation.

2.2: Our system architecture

We simply used folders stored on our computer as "abstract nodes" to model the system. We had n nodes named D1; D2; [...] DN. The last two folders (DN-1 and DN) represent the p and q parity disks. The previous N-2 disks are the data storage disks. For our implementation, we used four data stripes nodes and two parity nodes, for a total of six nodes. It is possible to run our program with a different number of storage nodes, it will then change the nbOfDirectories variable by assigning it the number that storage node you have. However, our system only works for two parity disks, p and q which will always be the last two folders DN-1 and DN (**figure 3**).

2.3: Data Storage

For data storage some assumptions have been made. First, the storage nodes are simple folders which contain data chunks. The data chunks are text files of different sizes. The nodes have the same storage capacity and no

limit assumed. We apply a Vertical stripe and always store our parity in the last 2 nodes.

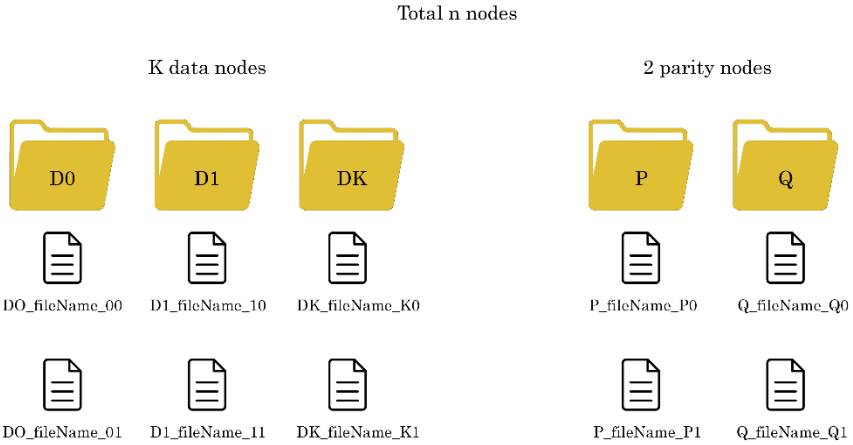

Figure 2: our system representation with folders as abstract nodes

In order to store a file in our distributed storage system, we must first break the file into Bytes: we read the file byte by byte, and we store each of the values in a table. Then simply distribute this file between the different storage disk we have (N-2 disks). Equally-sized segments are created and stored in the data disks/nodes. Each file is named in a specific way, using the name of the original file, in order to be able to find it easily.

The next steps are to create the parities. They are generated based on the segments and stored in the last 2 nodes/disks. To calculate p, it is the same as RAID-5 computation. Thus, it is a XOR operation between the different pieces of the files stored in the storage folders (algorithm 1). When a specific data block in the stripe is being updated, P value must be updating. For q, the calculation is a little more complicated, it is also necessary to multiply by 2 at each stage, as we can see in the **figure 4**. Same as P, when a specific data block in a stripe is being update, the Q value must also be update.

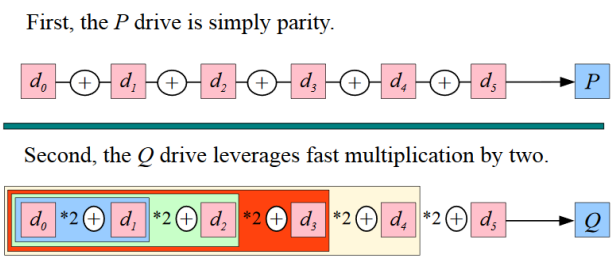

Figure 3: Equations for generate p and q from data stripes files. From [2] presentation.

P and Q elements are always written in the same two nodes but every modification of the files would require updating of theses nodes.

Algorithm 1: derive the value of p

```
Result: p value
input: A bitmap data which represent the files stores in the nodes
for i ← 1 to dataLength do
    for j ← 1 to data[i]Length do
        p[j] = p[j] XOR data[i][j];
    end
end
return p;
```

Algorithm 2: derive the value of q

```
Result: q value
input: A bitmap data which represent the files stores in the nodes
for i ← 1 to dataLength do
    for j ← 1 to data[i]Length do
        q[j] = q[j] x 2 XOR data[i][j];
    end
end
return q;
```

2.4: Access Files

After storing the file through the different folders, if a user wants to access his files, we needed to be able to reconstruct the different pieces of the file in order to recreate a single file, identical to the original file.

So, we go through the different folders, and we get the files for the file searched (as a reminder, we had named his files, so you can find them easily). We then concatenate

all these files; we translate the bytes into strings, and we can then recover the original file.

2.5: Failure Determination

The RAID-6 system can support the loss of one or two. In case of node loss, the system is able to reconstruct the lost data using the remaining nodes. Thus, a defective node detection system is required.

To do this, we list the nodes present in the system. If the number of files corresponds to what is expected (nbOfDirectories +2) all is well, otherwise, it means that it misses a folder, that one of the nodes fell. If this is the case, we look for what nodes are missing by checking their names (as a reminder, the nodes are named from 1 to N, where N is the total number of nodes). We can thus know which node is failing, to rebuild it afterwards.

It is also possible that a node is always present but that it does not work properly: it may be that a file is missing or that a file has been corrupted. These two cases are verified by our algorithm. In the case of a missing file, we check, for each node, if they are correctly filled. Because files are named according to a specific way, it is easy to know if a file is missing or not. In the case of a corrupted file, we check that its size is correct. If the file is too short, it means that part of it is missing.

2.6: Data recovery

In order to reconstruct the missing data, we considered several possible cases. With RAID-6, it is possible to recover data when two nodes fail simultaneously. To ensure redundancy, it is necessary to rebuild data that to was on the failed nodes from the remaining good nodes and rewrite it into the new nodes. There is different case to consider: node(s) containing P and/or Q fail; one data drive fail; P node and one data drive fail; Q node and one data drive fail; two data drive fail. It can't be another scenario, only these cases can happen in RAID-6. The rebuild steps for each case are presented below.

P and/or Q node failure

If the node containing p data and/or the one containing q failed, it is easy to recover it. Since all data drives are intact, it is enough to regenerate it using the same function that was used to generate it the first time (**algorithms 1 and 2**).

One data node failure

In this case, just a single data stripe node is faulty (the nodes containing p and q are correct). We then use p to find it. We could have used q as well, but since the equation is more complicated, it is easier to use p. Then just apply the XOR operation between the set of remaining nodes and p to find the missing file (**algorithm 3**).

Algorithm 3: recovering data node using p

Result: data value
input: *data* A bitmap which represent the files stores in the nodes;
nbOfDirectories an integer which represent the number of data node;
missingdir a tab where the num of missing directories are store ;
indexes Link between x of Dx in files tab;

```
for col ← 1 to data[0]Length do
    for j ← 1 to nbOfDirectories do
        pCoeff ← 1;
    end
    pResult ← 0;
    for d ← 1 to nbOfDirectories do
        if d not in missingdir then
            pResult ← pResult XOR pCoeff[d] x data[indexes[d]][col];
            pCoeff[d] ← O;
        end
    end
    file ← pResult XOR data[-1 - (nbOfDirectories + 1 not in
     missingDir][col]
end
return file;
```

P and one data node fail

In addition, we considered the case where two nodes are faulty. If it is the node containing p and a data stripe node (other than the one containing q), we used the value Q to rebuild the data node (**algorithm 4**). After rebuilding the fail data node, P value can be recomputed using a normal XOR operation (**algorithm 1**).

Q and one data node fail

If the node containing q and a data stripe node are faulty, the rebuild process is like RAID-5 recovery. Since P node is intact, the data block can be rebuilt using p value (**algorithm 3**). Then, since all data node value are correct, we can rebuild Q using **algorithm 2**.

Two data nodes fail

Finally, if two data nodes are faulty, it is the most difficult case. If node 1 and node 2 failed, we can write the

two equations from **figure 4** with two unknowns, Dx and Dy. We now have to solve this system of equation with two equation and two unknowns (**algorithm 5**)

Algorithm 4: recovering data node using q

```
Result: data value
input: data A bitmap which represent the files stores in the nodes;
nbOfDirectories an integer which represent the number of data node;
missingdir a tab where the num of missing directories are store ;
indexes Link between x of Dx in files tab;
for col ← 1 to data[0]Length do
    for j ← 1 to nbOfDirectories do
        qCoeff ← 2 power (nbOfDirectories − j − 1);
    end
    qResult ← 0;
    for d ← 1 to nbOfDirectories do
        if d not in missingdir then
            qResult ← qResult XOR qCoeff[d] x data[indexes[d]][col];
            qCoeff[d] ← O;
        end
    end
    file ← qResult XOR data[-1][col]
end
return file;
```

Algorithm 5: recovering two data node using p and q

```
Result: data value
input: data A bitmap which represent the files stores in the nodes;
nbOfDirectories an integer which represent the number of data node;
missingdir a tab where the num of missing directories are store ;
indexes Link between x of Dx in files tab;
for col ← 1 to data[0]Length do
    for j ← 1 to nbOfDirectories do
        qCoeff ← 2 power (nbOfDirectories − j − 1);
        pCoeff ← 1;
    end
    qResult ← 0;
    pResult ← 0;
    for d ← 1 to nbOfDirectories do
        if d not in missingdir then
            qResult ← qResult XOR qCoeff[d] x data[indexes[d]][col];
            pResult ← pResult XOR pCoeff[d] x data[indexes[d]][col];
            qCoeff[d] ← O;
            pCoeff[d] ← O;
        end
        if nbOfDirectories not in missingdir then
            pResult ← pResult XOR data[−2][col];
        end
        if nbOfDirectories + 1 not in missingdir then
            qResult ← qResult XOR data[−1][col];
        end
        firstFile, secondFile ← solve(missingDir, pResult,
          qResult,nbOfDirectories);
    end
end
return firstFile, secondFile;
```

3: Limitations

Storage

Our system assumes that data nodes have the same storage capacity; there is no upper bound limit as well. We can store as many files as needed equally. The system does not account for file changes in which case, the system would have to reassemble the file and regenerate the parities and data chunks.

4: Experiments

To check the performance of our algorithm, we measure how long it takes to generate parities and to recover files of different sizes. It tells us how long is necessary to recover data before tolerating another failure. We also measure how much space p and q are taking.
To do this, we used an online random large file generator to create the files and we measure the performance for each of these files. All the results have been reported in the tab in **figure 4.**

File size	Size of P	Size of Q	Time to Generate P	Time to Generate Q	Time to recover one file	Time to recover two files
1MB	406KB	853KB	1.01	0.88	1.52	7.9
1KB	1KB	1KB	0.025	0.02	0.023	0.04
6MB	3126KB	5075KB	6.1	5.7	9.6	53

Figure 4: results of the experiments (time in seconds)

3: Conclusion

A RAID 6 storage system provides enhanced data protection for critical user data. But it is challenging to implement a RAID 6 system due to its complexity. In this paper, an overview of RAID 6 theory and implementation was presented. We used folders on or computer to represent storage drives as well as we present the implementation, the way to store data, to access data, to generate parity node and to rebuild fail node is the same.

References

[1] Intelligent RAID 6 Theory Overview and Implementation, Intel Corporation

[2] Tutorial on Erasure Coding for Storage Application, Part 1, James S. Plank, February 12, 2013